Programming Techniques and Data Structures

M. Douglas McIlroy Editor

An On-Line Algorithm for Fitting Straight Lines Between Data Ranges

Joseph O'Rourke
The Johns Hopkins University

Applications often require fitting straight lines to data that is input incrementally. The case where a data range $[\alpha_k, \omega_k]$ is received at each t_k, $t_1 < t_2 < \ldots t_n$, is considered. An algorithm is presented that finds all the straight lines $u = mt + b$ that pierce each data range, i.e., all pairs (m, b) such that $\alpha_k \le mt_k + b \le \omega_k$ for $k = 1, \ldots, n$. It may be that no single line fits all the ranges, and different alternatives for handling this possibility are considered. The algorithm is on-line, producing the correct partial result after processing the first k ranges for all $k < n$. For each k, the set of (m, b) pairs constitutes a convex polygon in the m-b parameter space, which can be constructed as the intersection of $2k$ half-planes. It is shown that the $O(n \log n)$ half-plane intersection algorithm of Shamos and Hoey can be improved in this special case to $O(n)$.

Key Words and Phrases: on-line algorithms, incremental algorithms, computational geometry, computational complexity, half-plane intersection, piecewise linear approximation

CR Categories: 3.63, 4.49, 5.13, 5.25, 5.39

This work was supported in part by an IBM Graduate Research Fellowship and the National Science Foundation under Grant MCS78-07466 while the author was a student at the University of Pennsylvania.

Author's Present Address: Joseph O'Rourke, Electrical Engineering Department, The Johns Hopkins University, Baltimore, MD 21218.

I. Introduction

It is frequently useful to find straight lines that are consistent with a set of n data ranges. This problem can be reduced to that of intersecting $2n$ half-planes, a computation which Shamos and Hoey have shown to be achievable in $O(n \log n)$ time [6]. It will be shown that the ordering of the data inherent in the line-fitting problem allows a special purpose algorithm to operate in $O(n)$ time, which is optimal for this problem.

Consider an incremental process that produces data at successive discrete values of a variable t. If the data consists of a range $[\alpha_k, \omega_k]$ in the variable u at each t_k point, then the process can be characterized by the stream of triples $(t_1, \alpha_1, \omega_1)$, $(t_2, \alpha_2, \omega_2)$, $\cdots$. Frequently the variable t will represent time, but any variable whose successive values are monotonically increasing will serve as well. The extent of the range may represent the estimated error of some measurement technique, or directly determined bounds on the variable u. In any case, the range is interpreted as signifying all the valid values of u at a particular t; values outside of the range are considered invalid or impossible. Under these conditions, it may be useful to approximate the data up to a certain t_n point by linear functions, both for analysis of the past data and for prediction of future performance.

For real-time applications, it is important that an algorithm be *incremental*. For the problem under consideration, this means that a fit for k data ranges can be derived from a fit for the first $k - 1$ ranges by an incremental computation. An algorithm that produces its output in this fashion is called *on-line* [7]. This requirement rules out the possibility of employing any type of backtracking to search for the optimum "knot" points, as is done, for example, in split–merge algorithms [4].

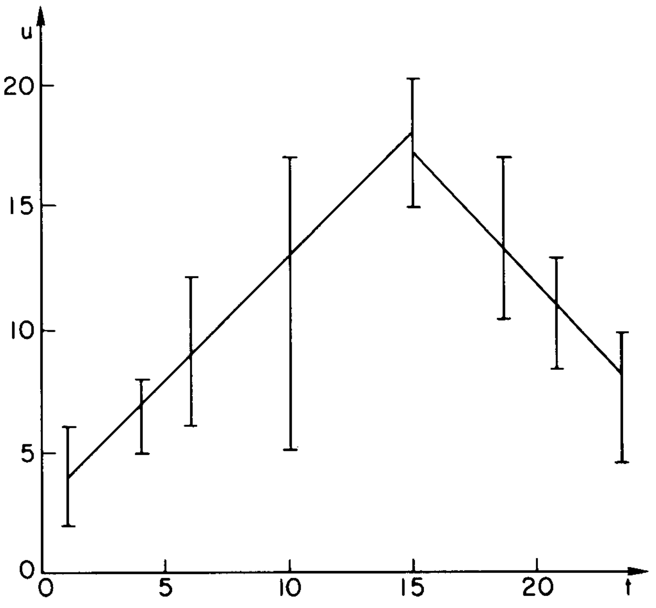


Fig. 1. A Number of Data Ranges in the Variable u at Various Values of t. The first five ranges can be pierced by a number of straight lines; one such is shown. No single straight line can fit the first six ranges, however, and so a new linear piece is initiated.

Rather a straightforward serial computation is called for: computing the set of lines that fit the ranges up to time point t_{k-1}, then modifying this set to include the data at t_k, and so on. This methodology is adopted here by terminating a linear piece whenever the next data range is inconsistent with a single linear approximation (an alternative scheme is considered in Sec. VI). Figure 1 illustrates this: although the first five ranges can be fit by a number of lines, no straight line can fit the first six ranges, so the fitting computation is started anew after the fifth range. Similar schemes have been used with linear least-squares approximations to achieve real-time behavior [1].

The set of all straight lines $u = mt + b$ that fit the n data ranges $(t_k, [\alpha_k, \omega_k])$, $k = 1, \ldots, n$ can be represented as the set of all (m, b) pairs that satisfy the equations

$$\alpha_k \leq mt_k + b \leq \omega_k, \tag{1}$$

for $k = 1, \ldots, n$. We will call this set of (m, b) pairs P_n.

II. Half-Plane Intersection

The set of equations (1) can be viewed as the constraint equations of a two-variable (m and b) linear-programming problem, and the set P_n as the "feasible region" in the m–b parameter space. Each equation represents two half-planes with parallel edges in the m–b space: the constraint at t_k corresponds to the half-planes

$$\begin{aligned} b &\geq (-t_k)m + \alpha_k \\ b &\leq (-t_k)m + \omega_k. \end{aligned} \tag{2}$$

The region P_n can thus be constructed as the intersection of the $2n$ half-planes arising from the n equations. P_n is therefore a convex polygon of at most $2n$ edges and $2n$ vertices. For example, the first five data ranges shown in Figure 1 generate the convex polygon of six edges shown in Figure 2. Any point inside or on the boundary of P_n could serve as a representative fitting line; for example, the center of gravity of the polygon's vertices will be an internal point, as will the mean of any three vertices.

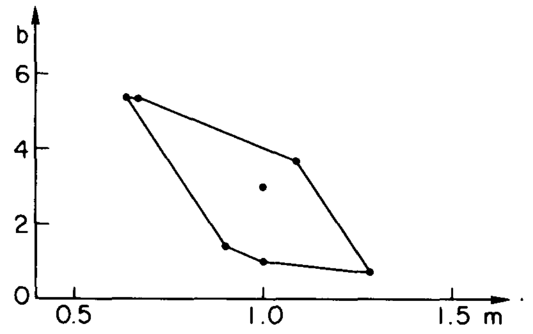


Fig. 2. The Convex Polygon in m–b Parameter Space Corresponding to the First Five Data Ranges of Figure 1. The polygon has six edges and six vertices. Every point inside the polygon represents a straight line that fits the five data ranges. The single interior point shown corresponds to the straight line drawn in Figure 1.

The problem of computing the intersection of half-planes has been studied by Shamos and Hoey [6]. The most straightforward algorithm requires $O(n^2)$ time for n half-planes, but Shamos and Hoey show the intersection may be found in $O(n \log n)$ time. Although their algorithm uses a "divide-and-conquer" technique, and so does *not* produce $P_1, P_2, \ldots, P_{n-1}$ in the process of computing P_n as we require, their approach can be modified to yield an $O(n \log n)$ algorithm for our problem.

It is the main purpose of this paper to show that this time complexity can be improved by taking advantage of the special circumstances of the line-fitting problem. Exploiting the inherent ordering of the data permits the construction of an $O(n)$ algorithm.[1]

III. Polygon Properties

It is necessary to establish some simple properties of the polygons used by the algorithm. It is assumed throughout that all time points are positive and are received in order: $t_k > 0$ and $t_k < t_{k+1}$ for all k.

LEMMA 1. *Each edge of each polygon P_k in m–b parameter space has a strictly negative slope. In fact, using the notation of Eq.* (1), *the slope of each edge of P_k lies in the range* $[-t_k, 0)$.

PROOF. Polygon P_k is the intersection of $2k$ half-planes, and so each edge of P_k lies along one of the $2k$ half-plane edges. Each half-plane is described by one of the forms displayed as Eqs. (2) above, and therefore each half-plane edge has a slope of $-t_i$. By the assumptions that $t_i > 0$ and $t_i < t_k$ for $1 \leq i < k$, each slope lies within $[-t_k, 0)$. □

LEMMA 2. *For each polygon P_k in m–b parameter space, the rightmost point of the polygon is also the lowest point, and the leftmost point is also the highest point. More precisely, define*

$$mmax_k = \max\{m : (m, b) \text{ is a vertex of } P_k\}$$
$$bmax_k = \max\{b : (m, b) \text{ is a vertex of } P_k\}$$

and define $mmin_k$ and $bmin_k$ similarly. Then $(mmax_k, bmin_k)$ and $(mmin_k, bmax_k)$ are both vertices of P_k.

PROOF. Suppose the lemma is not true. Then (to take one-half of the lemma), the rightmost and the lowest points are distinct vertices. This means that there exists a m' and a b' such that $m' < mmax_k$ and $b' > bmin_k$ with $(mmax_k, b')$ and $(m', bmin_k)$ both vertices of P_k. This situation is illustrated in Figure 3. It is clear from the figure that it is not possible to connect the lowest and rightmost vertices together to close off the lower right end of the polygon without including at least one edge of positive slope. By Lemma 1, however, all of the polygon edges have a negative slope. A similar contra-

[1] See [2] for another special case improvement of the $O(n \log n)$ half-plane intersection algorithm.

Fig. 3. An Impossible Polygon Illustrating the Proof of Lemma 2. If the lowest and rightmost vertices of the polygon are distinct, as they are in this figure, then an edge of positive slope (such as a) is needed to close off the lower right portion of the polygon. This is not possible, since by Lemma 1, all polygon edges have a negative slope.

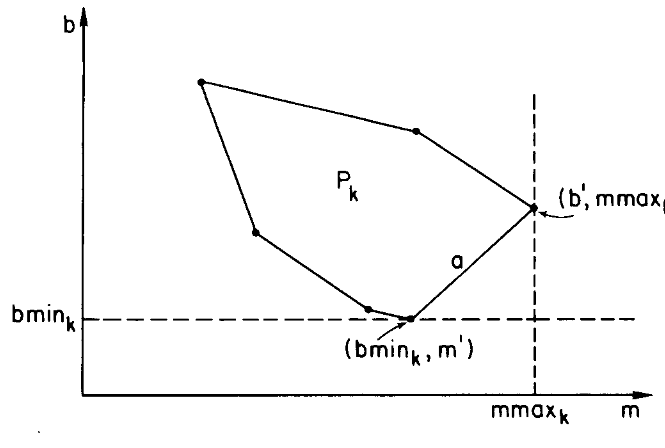
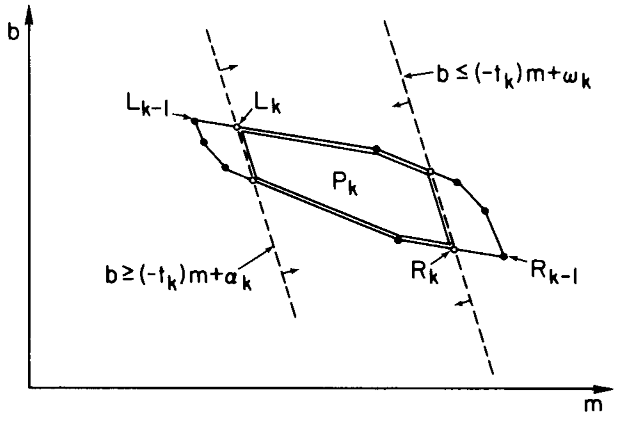


Fig. 4. Polygon P_k is Formed by Intersecting Polygon P_{k-1} with the Two Half-Planes Derived from the kth Data Range. The search for the intersection points between polygon P_{k-1} and the right half-plane edge starts at R_{k-1} and moves towards L_{k-1} along the upper and lower polygon halves; for the left half-plane edge, the search proceeds from L_{k-1} to R_{k-1}. The points falling outside of the half-planes are discarded, and polygon P_k is what remains.



diction arises if the leftmost and highest points are assumed to be distinct. □

This last lemma implies that the polygon fits into a rectangle whose sides are parallel to the m and b axes. For each polygon P_k, call the upper left corner point of this rectangle L_k and the lower right corner point R_k. These two points divide the polygon edges into an "upper" half (all those above the rectangle diagonal between L_k and R_k) and a lower half (all those below), and play a key role in the algorithm that follows.

IV. The Algorithm

The algorithm for computing P_k from P_{k-1} searches sequentially from the corner points for the points at which the half-plane edges intersect P_{k-1}. Since P_{k-1} can have $2(k-1)$ vertices, this search may require $O(k)$ time for any particular k. However, the search will not be $O(k)$ for *all* k, $1 \le k \le n$, since each extra vertex examined at step k is deleted by the intersection and so is not looked at in later steps. The end result is that the total work to compute $P_1, P_2, \ldots, P_n$ is only $O(n)$. This is proven after a precise statement of the algorithm is given.

It is assumed in the algorithm that a polygon is represented as a doubly linked circular list of its vertices.

```
Algorithm
Input: $(t_i, \alpha_i, \omega_i)$, $i = 1, 2, \ldots, n$
Computed: polygons $P_1, P_2, \ldots, P_n$
begin {algorithm}
Compute $P_2$ and initialize $L_2$ and $R_2$.
for $k = 3$ to $n$ do
begin {for loop}
  if $R_{k-1}$ is to the left of the half-plane $b \ge (-t_k)m + \alpha_k$ or
     $L_{k-1}$ is to the right of the half-plane $b \le (-t_k)m + \omega_k$
    then {null intersection}
      begin {reinitialize}
        Set $P_k$ to the quadrilateral formed from the data ranges at $t_{k-1}$
        and $t_k$.
        Initialize $L_k$ and $R_k$.
      end {reinitialize}
    else {non-null intersection}
      begin {intersect}
        (1) Search for the two intersection points of the half-plane
            $b \le (-t_k)m + \omega_k$ with $P_{k-1}$ as follows (refer to Figure 4):
            (a) Compare the vertex $R_{k-1}$ with the half-plane edge.
                If $R_{k-1}$ falls inside of the half-plane, then the half-
                plane does not clip the polygon at all; in this case,
                go to step (2).
            (b) Examine in turn each edge of the polygon $P_{k-1}$
                from $R_{k-1}$ to $L_{k-1}$ along the *upper* half of the
                polygon until an intersection is found. More specif-
                ically, for each vertex along the upper half, starting
                with the vertex immediately to the left of $R_{k-1}$,
                check whether or not it falls inside the half-plane.
                If it does, then the half-plane edge intersects the
                polygon edge whose left end point is that vertex.
            (c) Examine in turn each edge of the polygon $P_{k-1}$
                from $R_{k-1}$ to $L_{k-1}$ along the *lower* half of the
                polygon until an intersection is found by comparing
                the vertices against the half-plane as in (b) above.
                Set $R_k$ to the intersection point.
        (2) Search for the two intersection points of the half-plane
            $b \ge (-t_k) + \alpha_k$ with $P_{k-1}$ with steps similar to (1a), (1b),
            and (1c) above, except conduct all searches *from* $L_{k-1}$ *to*
            $R_{k-1}$, and update the point $L_k$.
        (3) Remove from the polygon $P_{k-1}$ all those vertices outside
            of the two intersecting half-planes, adding in the new
            vertices caused by the intersection, as illustrated in Figure
            4. Set $P_k$ to the resulting polygon.
      end {intersect}
end {for loop}
end {algorithm}
```

This algorithm has been implemented and used in a computer vision system to predict the positions of moving objects in successive images [3]. An example of the algorithm's performance is shown in Figure 5. Here a representative line (corresponding to a point inside of P_k) is displayed at each t_k. Note that there was a single reinitialization, at $t_5 = 0.8$.

Fig. 5. Representative Lines From the Polygons Constructed by the Algorithm.

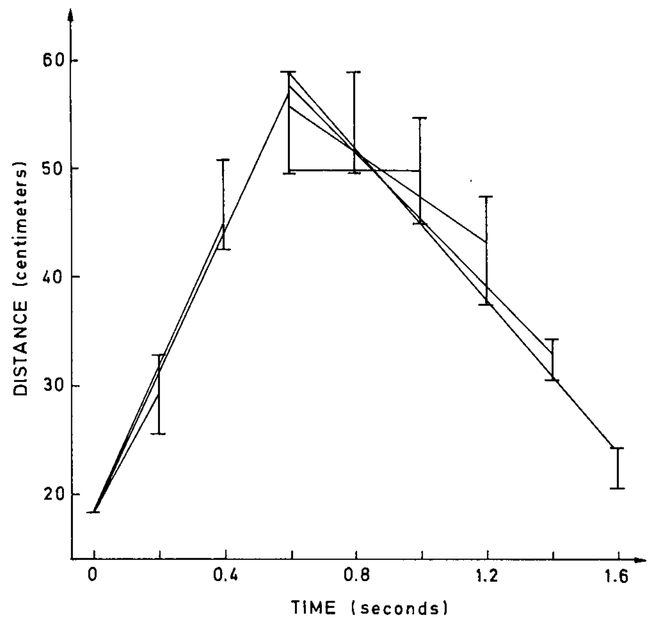
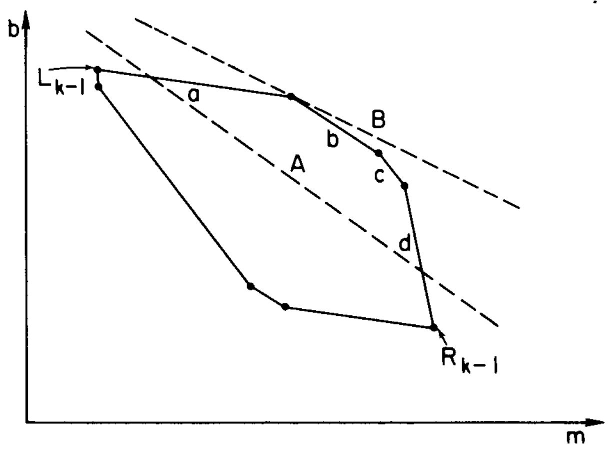


Fig. 6. Two Impossible Half-Plane Edges Illustrating the Proof of Theorem 1. Edge *A* intersects the upper half of the polygon in two places, and although its slope is steeper than polygon edge *a*, it is shallower than *d*. Edge *B* just grazes the polygon at a single point, and its slope is steeper than *a* but shallower than *b*. Therefore both *A* and *B* are impossible, since a half-plane edge arising from the *k*th data range must have a steeper slope than each edge of P_{k-1}.



V. Correctness and Complexity

The following two theorems establish that the algorithm is correct and has a time complexity of $O(n)$.

THEOREM 1. *The algorithm correctly computes the sequence of polygons* $P_1, P_2, \ldots, P_n$.

PROOF. Each polygon P_k is the intersection of $2k$ half-planes of the form shown in Eqs. (2). Thus the algorithm is correct if it computes these intersections properly. Establishing this requires showing that: (1) each half-plane edge either does not intersect the polygon at all, or intersects the upper half once and the lower half once (where L_k and R_k are considered members of both halves, so that if the edge intersects only at one of these points, it is considered to intersect both halves); and (2) the algorithm correctly computes the intersection points. Assuming the truth of (1), it is clear from the design of the algorithm that (2) is satisfied, and so only (1) will be proved.

Suppose polygon P_{k-1} has been constructed, and the half-plane edge $b = (-t_k)m + \omega_k$ intersects P_{k-1}. From Lemma 1, all the edges of P_{k-1} have slopes in the range $[-t_{k-1}, 0)$. Since the half-plane edge has slope $-t_k$ and since $t_k > t_{k-1}$, the half-plane edge is steeper than any of the polygon edges. This implies that it may only cut the upper half of the polygon at a single point; to intersect it in two places would require the half-plane edge to have a shallower slope than either the polygon edge cut on entrance to the polygon or an edge cut on exit (see line A in Figure 6), and intersecting in more than two places would contradict convexity of the polygon. If the single intersection point on the upper half is not L_{k-1} or R_{k-1}, then the half-plane edge must enter the polygon; a grazing edge (such as B in Figure 6) must have a shallower slope than some polygon edge. Therefore the half-plane edge intersects the lower half of the polygon. Similar reasoning shows that the half-plane edge may only intersect the lower half of the polygon at a single point. □

THEOREM 2. *The time complexity of the algorithm is* $O(n)$, *where n is the number of input data ranges.*

PROOF. The time complexity of the algorithm is determined by the number of vertex-to-half-plane comparisons made. [Step (3) of the algorithm can be accomplished by manipulating a fixed number of pointers.] The worst case occurs when no reinitialization (caused by a null intersection) takes place throughout the entire construction from 1 to *n*, and this will be assumed in the analysis below.

Let H_k be the number of half-plane comparisons made at iteration *k* of the **for** loop of the algorithm. Then the order of the algorithm is the order of

$$\sum_{k=3}^{n} H_k.$$

Let D_k be the number of vertices deleted by the algorithm at step *k*, i.e., $D_k = |P_{k-1}| - |P_k|$, where $|P_k|$ is the number of vertices of polygon P_k. Note that D_k may be negative, indicating a net gain of vertices. A relationship will be established between H_k and D_k that will provide an upper bound on the complexity.

Within one iteration of the **for** loop, step (1a) of the algorithm makes one half-plane comparison. If steps (1b) and (1c) are reached, and if they make h_b and h_c comparisons, respectively, they result in a net deletion of $h_b + h_c - 3$ vertices. In the example of Figure 4, step (1b) makes $h_b = 3$ comparisons, and step (1c) makes $h_c = 1$ comparison, and 3 vertices are deleted and 2 added for a net deletion of 1 vertex. Thus, in the case where the half-plane edge does intersect the polygon, a total of $h_1 = 1 + h_b + h_c$ comparisons are made in step (1) and $d_1 = h_b + h_c - 3 = h_1 - 4$ vertices are deleted. If there is no intersection, and step (1a) exits, then a single comparison is made and no vertices deleted: $h_1 = 1$ and $d_1 = 0$. In either case, it is true that $d_1 \geq h_1 - 4$.

Similar reasoning shows that the relationship between h_2, the number of half-plane comparisons made in step (2) of the algorithm, and d_2, the number of vertices deleted by that step, is likewise $d_2 \geq h_2 - 4$. Since at each step $D_k = d_1 + d_2$ and $H_k = h_1 + h_2$, we have $H_k - 8 \leq D_k$.

Now since the maximum number of vertices resulting from the intersection of $2n$ half-planes is $2n$, it must be the case that

$$\sum_{k=3}^{n} D_k \leq 2n,$$

and therefore

$$\sum_{k=3}^{n} (H_k - 8) \leq 2n \quad \text{or} \quad \sum_{k=3}^{n} H_k \leq 10n - 16.$$

Therefore the order of the algorithm is $O(n)$. □

Note that this order is optimal, since it is necessary to at least examine each of the n input data ranges, and this will always require $O(n)$ time. Also note that the worst case space complexity is $O(n)$, since storage is only needed for the vertices of the polygon, which may have as many as $2n$ vertices.

VI. Discussion

Since the algorithm is $O(n)$ for n inputs, its average update time *per input* is $O(1)$. However, it is clear that one particular update may be $O(n)$, or more precisely, $O(r)$, where $r \leq n$ is the length of the longest "run," i.e., the longest stretch between reinitializations. In order for the algorithm to be *real-time*, the update time per step must be less than the interarrival delay between successive inputs [5]. If the runs can be arbitrarily long, then real-time can not be achieved with the proposed algorithm (assuming uniform interarrival delay); if the data is sufficiently "choppy" to bound r by a constant, however, real-time could be achieved. In any case, the algorithm could be used for real-time applications as long as a delay of $O(r)$ in output production could be tolerated.

Finally, we note that by changing the sign of each t_i and shifting the origin, the same algorithm could be used to construct a polygon representing the lines that fit the ranges at $t_n, t_{n-1}, \ldots, t_k$, working backwards rather than forwards. If, at each t_n, this construction is carried backwards as far as possible before the polygon becomes null, the result gives an estimate of the linear trend at t_n. Although this computation might be more useful for some applications than the piecewise approximation produced by the algorithm presented in Sec. IV, it is, in general, more costly, possibly involving $O(n)$ computation at each step.

Acknowledgments. I would like to thank G. Radack for a careful reading of the manuscipt, and the referees for many valuable suggestions.

Received 8/79; revised 2/80; accepted 4/81

References

1. Freedman, A.M., Buneman, O.P., Peckham, G., and Trattner, A. Automatic recognition of significant events in the vital signs of neonatal infants. *Comput. Biomed. Res. 12,* 2 (Apr. 1979), 141–148.
2. Lee, D.T., and Preparata, F.P. An optimal algorithm for finding the kernel of a polygon. *J. ACM 26,* 3 (July 1979), 415–421.
3. O'Rourke, J., and Badler, N. Model-based image analysis of human motion using constraint propagation. *IEEE Trans. on Pattern Analysis and Machine Intelligence, PAMI-2,* 6 (Nov. 1980), 522–536.
4. Pavlidis, T. *Structural Pattern Recognition.* Springer-Verlag, Berlin, 1977, pp. 168–184.
5. Preparata, F.P. An optimal real-time algorithm for planar convex hulls. *Comm. ACM 22,* 7 (July 1979), 402–405.
6. Shamos, M.I., and Hoey, D. Geometric intersection problems. 17th Ann. IEEE Symp. Foundations of Computer Science, (Oct. 1976), 208–215.
7. Shamos, M.I. Computational geometry. PhD Dissertation, Yale University, (1978), p. 62.

Corrigendum: Operating Systems

Glenn Ricart and Ashok K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," *Comm. ACM 24,* 1 (Jan. 1981) 9–17.

Kiyoshi Ishihata of the University of Tokyo has pointed out a local, intranode synchronization fault in the published version of our mutual exclusion algorithm [1]. We had corrected the same difficulty early in 1980 [2] but mistakenly provided an older version for publication.

When the receiver process does not act as an interrupt process to the main process, it is possible that the receiver is deciding to defer a message while the main process is simultaneously exiting from its critical section. The present arrangement of the shared variable semaphore must be strengthened.

In the main process, protect the critical section exit:

```
P (Shared_vars);
  Requesting_Critical_Section := FALSE;
V (Shared_vars);
```

In the receiver process, extend the protection now ending after the assignment to Defer__it to include the following **IF** statement:

```
   Defer_it := Requesting_Critical_Section AND . . .
IF Defer_it THEN Reply_Deferred[j] := TRUE ELSE
        Send_Message (REPLY, j);
V  (Shared_vars);
```

References

1. Ricart, G. and Agrawala, A. K. An optimal algorithm for mutual exclusion in computer networks. *Comm. ACM 24,* 1 (Jan. 1981), 9–17.
2. Ricart, G. Efficient synchronization algorithms for distributed systems. Univ. of Maryland Tech. Rept. TR-902, May 1980.